Exploiting Intermediate Sparsity in Computing Derivatives for a Leapfrog Scheme*

Christian H. Bischof and Po-Ting Wu
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439-4844
e-mail: {bischof,pwu}@mcs.anl.gov

Argonne Preprint ANL/MCS-P572-0396

Abstract

The leapfrog scheme is a commonly used second-order method for solving differential equations. Letting Z denote the state of the system, we compute the state at the next time step as $Z(t+1) = H(Z(t), Z(t-1), W)$, where t denotes a particular time step, H is the nonlinear timestepping operator, and W are parameters that are not time dependent. In this article, we show how the associativity of the chain rule of differential calculus can be used to expose and exploit intermediate derivative sparsity arising from the typical localized nature of the operator H. We construct a computational harness that capitalizes on this structure while employing automatic differentiation tools to automatically generate the derivative code corresponding to the evaluation of one time step. Experimental results with a 2-D shallow water equation model on IBM RS/6000 and Sun SPARCstations illustrate these issues.

Keywords: Automatic Differentiation, Sparsity, Jacobian, Leapfrog Scheme.

*This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Computational and Technology Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

1 Introduction

The leapfrog method is a commonly used second-order method for solving differential equations (see, for example, [11, pp. 53 ff.]). Let $Z(t)$ and $Z(t-1)$ denote the current and previous state of a time-dependent system, respectively. Except for possibly the initial steps, a leapfrog scheme computes the state $Z(t+1)$ at the next time step as

$$Z(t+1) = H(Z(t), Z(t-1), W), \tag{1}$$

where H is the nonlinear operator advancing the system state. and W are system parameters that are not time-dependent. That is, the code has the structure shown in Figure 1.

```
Initialize Z(0) and W.
Compute Z(1).
for t = 1 to T − 1 do
    Z(t + 1) = H(Z(t), Z(t − 1), W);
end do
```

Figure 1: Schematic of a Leapfrog Scheme

In applications such as parameter identification one may be interested in $\frac{d\,Z(T)}{d\,X}$, where T is the final time step and X denote the variables whose derivatives one is interested in, typically a subset of the initial inputs of the model. Using automatic differentiation (AD) tools, we can easily generate derivative code to compute $\frac{d\,Z(T)}{d\,X}$. A discussion of the various approaches to AD can be found in [2, 10] and an overview of currently available AD tools at URL `http://www.mcs.anl.gov/Projects/autodiff/AD_Tools`. These tools will generate code that computes accurate derivatives and it will be reasonable efficient given the fact that *no knowledge of the underlying problem was used in generating the derivative code.* As this article will show, however, we can favorably exploit the structure in typical PDE-based computations arising from the sparsity of the underlying operators in the use of AD tools.

To illustrate, we let $Z \in \mathbb{R}^n, W \in \mathbb{R}^p$, and let $X \in \mathbb{R}^s, X \subseteq [Z(0), W], s \leq n+p$, denote the variables with respect to which we are differentiating. Tools such as ADIFOR [5] or ADIC [8], or any other tool that is based mainly on the so-called forward mode of automatic differentiation can compute $\frac{d\,Z(T)}{d\,X}$ at a cost on the order of s times that of just computing $Z(T)$ (in terms of both runtime and memory) by maintaining derivatives of program variables with respect to X.

To motivate how we can do better, we differentiate (1) and obtain

$$\frac{d\,Z(t+1)}{d\,X} = \frac{\partial\,H}{\partial\,Z(t)} \cdot \frac{d\,Z(t)}{d\,X} + \frac{\partial\,H}{\partial\,Z(t-1)} \cdot \frac{d\,Z(t-1)}{d\,X} + \frac{\partial\,H}{\partial\,W} \cdot \frac{d\,W}{d\,X}, \tag{2}$$

where all partial derivatives of H are evaluated at $(Z(t), Z(t-1), W)$. Typically, the matrices $\frac{\partial\,H}{\partial\,Z(t)}$ and $\frac{\partial\,H}{\partial\,Z(t-1)}$ are sparse due to the local nature of the stencil employed in H, and one can exploit this fact to compute them inexpensively using forward-mode based AD tools [1, 7, 9]. Thus, if we use the aforementioned AD tools to (cheaply) compute $\frac{\partial\,H}{\partial\,Z(t)}$ and $\frac{\partial\,H}{\partial\,Z(t-1)}$, and then form $\frac{d\,Z(t+1)}{d\,X}$ through a series of matrix-matrix multiplications, we may well come out ahead.

This article is structured as follows. In the next section, we review the capabilities of current forward-mode-based AD tools with respect to computing arbitrary directional derivatives and show how chain rule associativity allows us to exploit intermediate sparsity in the problem. In section 3, we introduce a 2-D shallow water equation model problem [12, 13]. In Section 4, we present experimental results obtained with this code, using ADIFOR-generated derivative code on an IBM RS/6000 and Sun SPARCstation platforms. Lastly, we summarize our results.

2 Stencil-Induced Intermediate Sparsity

Automatic differentiation (AD) is a technique for augmenting codes with statements for the computation of derivatives. Compared with other approaches for computing derivatives (e.g., by hand, divided differences, or symbolic approaches), it provides guaranteed accuracy, ease of use, and computational efficiency. The ADIFOR [4, 5] and ADIC [8] tools for Fortran and ANSI-C, respectively, directly generate derivative code (in Fortran or ANSI-C) that, given a user's specification of dependent and independent variables, computes the partial derivatives of all of the specified dependent variables with respect to all of the specified independent variables, in addition to the original result. As discussed in [5], the so-called forward mode of automatic differentiation that underlies these tools allows not only the computation of the full Jacobian J associated with the chosen dependent and independent variables, but any set of directional derivatives $J \cdot S$.

Examples of the resulting flexibility in the use of the generated derivative code are shown in [6], but one particular use is that of exploiting sparsity. For example, to compute a Jacobian J with the following structure (symbols denote nonzeros, and zeros are not

shown),

$$J = \begin{pmatrix} \bigcirc & & & \\ \bigcirc & & & \diamond \\ & \triangle & & \diamond \\ & \triangle & \square & \\ & \triangle & \square & \end{pmatrix},$$

we can exploit the fact that columns 1 and 2 and columns 3 and 4 have nonzeros in disjoint row positions. By initializing S as

$$S = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{pmatrix}.$$

we compute a so-called compressed version of J at roughly half the cost compared with that induced by setting S to a 4×4 identity. In general, the groups of columns that can be grouped together can be identified via a graph-coloring approach. This "compressed Jacobian" approach has been used successfully in large-scale optimization [1, 7].

An alternative approach to exploiting sparsity is to employ sparse data structures for the derivative objects in the AD-generated code. The SparsLinC library, which is integrated with the ADIFOR and ADIC tools, provides this functionality, and has been successfully employed in large-scale optimization [3, 7].

Let us now assume that, using a forward-mode-based AD tool, we have generated from the original code shown in Figure 1 a new code to compute derivatives of $Z(T)$ with respect to $Z(0)$ and W. A schematic of this code is shown in Figure 2. We call this the "black-box approach."

Initialize $Z(0)$, W, $\frac{dZ(0)}{dX}$, and $\frac{dW}{dX}$.
Compute $[Z(1), \frac{dZ(1)}{dX}]$.
for $t = 1$ **to** $T - 1$ **do**
$[Z(t+1), \frac{dZ(t+1)}{dX}] = g\_H(Z(t), \frac{dZ(t)}{dX}, Z(t-1), \frac{dZ(t-1)}{dX}, W, \frac{dW}{dX})$;
end do

Figure 2: Schematic of an Automatically Differentiated Leapfrog Scheme

In particular, a derivative-augmented version g_H of H was generated:

$$[Z(t+1), g\_Z(t+1)] = g\_H(Z(t), g\_Z(t), Z(t-1), g\_Z(t-1), W, g\_W). \qquad (3)$$

The g_H routine computes both $Z(t+1)$ as in (1) and

$$g\_Z(t+1) = \frac{\partial H}{\partial Z(t)} \cdot g\_Z(t) + \frac{\partial H}{\partial Z(t-1)} \cdot g\_Z(t-1) + \frac{\partial H}{\partial W} \cdot g\_W, \tag{4}$$

where all partial derivatives are evaluated at $(Z(t), Z(t-1), W)$.

The input derivative matrices $g\_Z(0)$ and g_W are usually referred to as "seed matrices," as their initialization determines what kind of (directional) derivatives is being computed. Let $I_{i\times j}$ and $0_{i\times j}$ denote an $i \times j$ identity and null matrix, respectively. If we were to initialize

$$g\_Z(0) = \left(\; I_{n\times n} \; , \; 0_{n\times p} \; \right) \text{ and } g\_W = \left(\; 0_{p\times n} \; , \; I_{p\times p} \; \right) \tag{5}$$

at the beginning of the code in Figure 2, upon completion we would have obtained

$$g\_Z(T) = \left(\; \frac{d\,Z(T)}{d\,Z(0)} \; , \; \frac{d\,Z(T)}{d\,W} \; \right)$$

at a cost that would be $O(n+p)$ times that of computing $Z(T)$ itself.

In general, let us assume that the computation of H takes f_H flops and $O(2*n+p)$ words of storage for every time step (typically, the state is only stored for the current and previous time step). Then, ignoring the startup cost in time step 0, for any subset X of s variables from the set of potential independent variables, a tool such as ADIFOR or ADIC can compute $\frac{d\,Z(T)}{d\,X}$ with $O(s*T*f_H)$ flops and $O(s*(2*n+p))$ words of storage overall.

In typical stencil-based computations, the timestepping operator H is of a localized nature: a particular gridpoint depends on its own value and the value of some of its neighbors at the current and previous time step. As a consequence, the Jacobians $\frac{\partial H}{\partial Z(t)}$ and $\frac{\partial H}{\partial Z(t-1)}$ will be sparse.

If we set

$$\begin{aligned} g\_Z(t) &= (I_{n\times n}, 0_{n\times n}, 0_{n\times p}) \\ g\_Z(t-1) &= (0_{n\times n}, I_{n\times n}, 0_{n\times p}) \\ g\_W &= (0_{p\times n}, 0_{p\times n}, I_{p\times p}) \end{aligned} \tag{6}$$

then the invocation of

$$g\_H\,(Z(t), g\_Z(t), Z(t-1), g\_Z(t-1), W, g\_W)$$

computes $2n+p$ derivatives and returns

$$g\_Z(t+1) = \left(\frac{\partial H}{\partial Z(t)}, \frac{\partial H}{\partial Z(t-1)}, \frac{\partial H}{\partial W} \right) .$$

If we exploited sparsity in a transparent fashion as is done with SparsLinC, for example, the cost of this computation will *not* be $2n + p$ times that of computing H, but rather κ that of computing H, where κ does not depend on n, but rather on the size of the stencil. In the sequel, we call this approach the "SparsLinC approach."

Alternatively, let S^1 and S^2 be suitably chosen seed matrices to compute compressed versions of $\frac{\partial H}{\partial Z(t)}$ and $\frac{\partial H}{\partial Z(t-1)}$ with λ_1 and λ_2 columns, respectively. Then we initialize

$$\begin{aligned} g\_Z(t) &= \left(S^1_{n\times\lambda_1}, 0_{n\times\lambda_2}, 0_{n\times p}\right) \\ g\_Z(t-1) &= \left(0_{n\times\lambda_1}, S^2_{n\times\lambda_2}, 0_{n\times p}\right) \\ g\_W &= \left(0_{p\times\lambda_1}, 0_{p\times\lambda_2}, I_{p\times p}\right). \end{aligned} \tag{7}$$

Upon return from g_H, the first λ_1 columns of $g\_Z(t+1)$ will contain a compressed version of $\frac{\partial H}{\partial Z(t)}$, the second λ_2 columns a compressed version of $\frac{\partial H}{\partial Z(t-1)}$, and the last p columns $\frac{\partial H}{\partial W}$. The cost for this approach is λ that of computing H, where $\lambda = \lambda_1 + \lambda_2 + p$. In the sequel, we call this approach the "compressed approach."

Independent of how $\frac{\partial H}{\partial Z(t)}$, $\frac{\partial H}{\partial Z(t-1)}$ and $\frac{\partial H}{\partial W}$ were computed, $\frac{dZ(t+1)}{dX}$ is then computed with three matrix-matrix multiplications using the already known quantities $\frac{dZ(t)}{dX}$, $\frac{dZ(t-1)}{dX}$, and $\frac{dW}{dX}$. The resulting algorithm is shown in Figure 3 and we call this approach the "intermediate sparsity" approach.

We note the following:

- Using Alternative 1, Step 1 requires $O(T * \kappa * f_H)$ flops and $O(\kappa * (2 * n + p))$ words of storage. Using Alternative 2, the cost of Step 1 is $O(T * \lambda * f_H)$ flops and $O(\lambda * (2 * n + p))$ words of storage. Typically, $\kappa \approx \lambda$, and the number of floating-point operations expended for the computation of H is proportional to n. Thus, the floating-point cost of computing the time-step derivatives scales with $T * n$, whereas the memory requirements are $O(n)$. This cost does not depend on the total number s of derivatives $\frac{dZ(T)}{dX}$ that we are computing. Independent of what overall derivatives are desired, we always compute derivatives with respect to $Z(t), Z(t-1)$, and W at every time step.
- Both $\frac{\partial H}{\partial Z(t)} \cdot \frac{dZ(t)}{dX}$ and $\frac{\partial H}{\partial Z(t-1)} \cdot \frac{dZ(t-1)}{dX}$ are, in general, multiplications of a sparse matrix of size $n \times n$ with a dense matrix of size $n \times s$. Thus, the cost of this step depends, in addition to n, on the number of nonzeros in $\frac{\partial H}{\partial Z(t)}$ and

Initialize $Z(0)$, W, $\frac{d\,Z(0)}{d\,X}$, and $\frac{d\,W}{d\,X}$.
Compute $[Z(1), \frac{d\,Z(1)}{d\,X}]$.
for $t = 1$ **to** $T - 1$ **do**

Step 1: Compute $\frac{\partial\,H}{\partial\,Z(t)}$, $\frac{\partial\,H}{\partial\,Z(t-1)}$, and $\frac{\partial\,H}{\partial\,W}$
via an invocation of g_H geared toward exploiting sparsity.

Step 2: Compute $\frac{d\,Z(t+1)}{d\,X}$ via a matrix-matrix multiplication:

$$\frac{d\,Z(t+1)}{d\,X} = \frac{\partial\,H}{\partial\,Z(t)} \cdot \frac{d\,Z(t)}{d\,X} + \frac{\partial\,H}{\partial\,Z(t-1)} \cdot \frac{d\,Z(t-1)}{d\,X} + \frac{\partial\,H}{\partial\,W} \cdot \frac{d\,W}{d\,X}$$

/* NOTE: This step assumes proper implementations of the multiplication of a sparse matrix by a dense matrix. */

end do

Figure 3: Schematic of a Differentiated Leapfrog Scheme Exploiting Intermediate Sparsity

$\frac{\partial\,H}{\partial\,Z(t-1)}$, the size n of Z, and the number s of derivatives in $\frac{d\,Z(T)}{d\,X}$ and its floating-point complexity scales with $T * n * s$. However, it does not depend on f_H.

- For this scheme to lead to computational improvements, the cost of the matrix-matrix multiply must not dominate the overall computational cost. This will be the case if sufficient effort is expended in the stencil computation (thus leading to a sizable f_H).
- The exploitation of intermediate sparsity only makes sense when κ and λ are small compared to s. In this case, we also expect some storage savings because the intermediate sparsity approach requires less storage for the derivatives of variables internal to H.

3 A 2-D Shallow Water Equation Model Problem

To illustrate the issues discussed previously, we employ a 2-D shallow water equations model problem [12, 13]. This code is simple, yet exhibits the characteristic features we based our algorithmic suggestions on, namely, a leapfrog method as time propagator, and a stencil-based propagation operator. In this particular example, any row

of $\left(\frac{\partial H}{\partial Z(t)}, \frac{\partial H}{\partial Z(t-1)}, \frac{\partial H}{\partial W}\right)$ has at most 13 nonzeros. The nonzero structure of this matrix is shown in Figure 4 for $n = 363$ and $s = 4$. The total number of nonzeros in the matrix is 3,101. As for $\frac{\partial H}{\partial Z(t-1)}$, it is essentially diagonal, with corner points attributing off-diagonal entries.

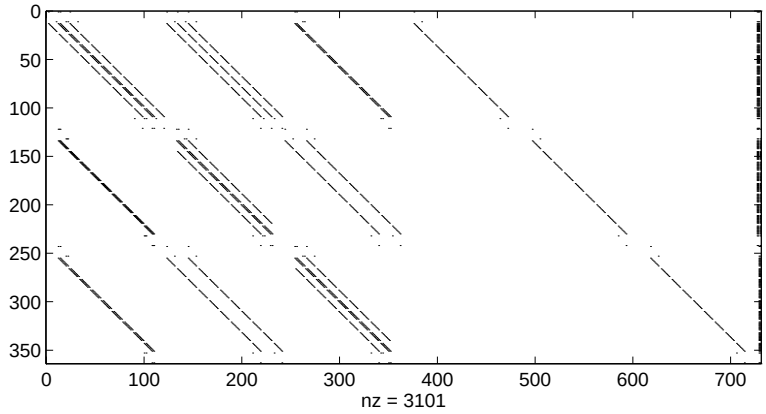

Figure 4: The Sparse Jacobian Associated with the Shallow Water Equations Model

Figure 5 shows the structure of the Jacobians $\left(\frac{dZ(t)}{dZ(0)}, \frac{dZ(t)}{dW}\right)$ for $t = 1, \ldots, 6$, using the derivative seeding (5). Note the fill-in as these sparse matrices are multiplied out. We noted that for $t > 9$ the structure of these Jacobians did not change any more and they were essentially full. Somewhat surprisingly, though, there were still some zero entries per row – the maximum number of nonzeros per row was 355 instead of $n + p = 367$ as one might have expected.

This fill-in is expected, as the stencil nature of the computation implies that information is spread further and further through the grid at every time step. The scheme outlined in Figure 3 concentrates this fill-in in the matrix accumulation step, while the derivative computation corresponding to a particular time step tries to exploit the sparsity shown in Figure 4. This phenomenon is further illustrated in Figure 6, which shows the average number of nonzeros occurring in derivative computations over the runtime of the code. The left plot shows the black-box approach with the seeding denoted in (5); the right plot shows the approach outlined in Figure 3. We see that in the black-box approach, the number of nonzeros steadily rises until it reaches 355. In contrast, the number of nonzeros in the derivative vectors in the approach that exploits the inter-

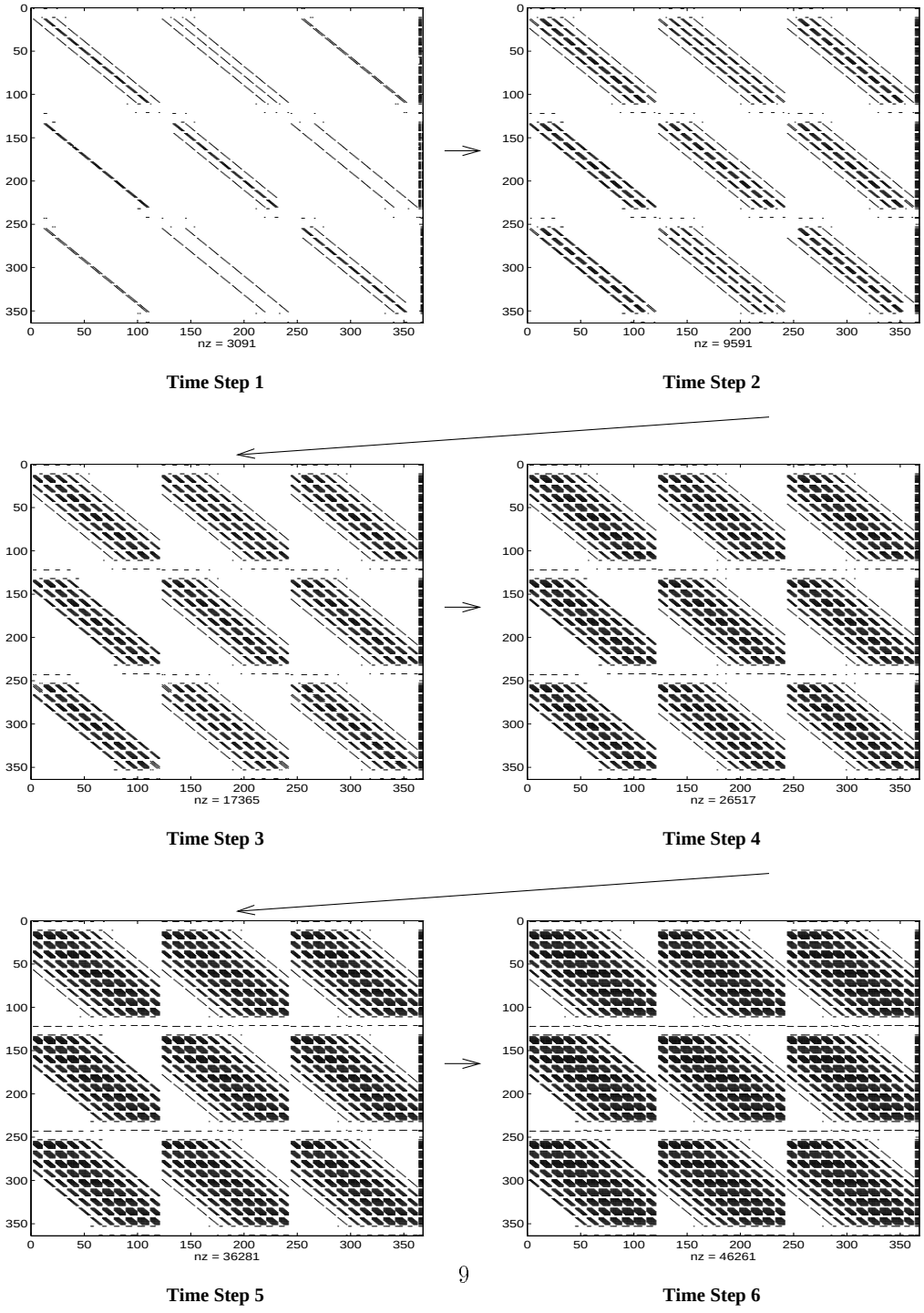

Figure 5: Jacobian Fill-in in the Shallow Water Equations Model

mediate sparsity is never more than 13, the maximum number of nonzeros per row of $\left(\frac{\partial H}{\partial Z(t)}, \frac{\partial H}{\partial Z(t-1)}, \frac{\partial H}{\partial W}\right)$, and structurally, except for the very first time step, the same computation takes place.

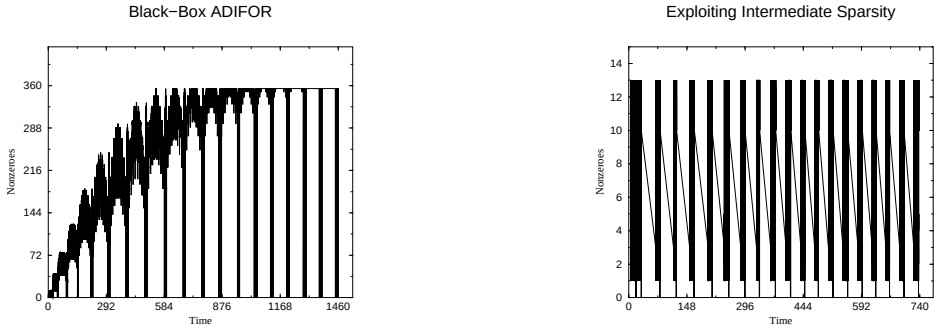

Figure 6: Nonzero Entries in Derivative Vectors over time in the "Black-Box" and "Intermediate Sparsity" Approach

4 Experimental Results

We compared the approach suggested with a "black-box" application of ADIFOR on Sun SPARCstation 5 and an IBM RS/6000 workstation platform. For the problems shown in Table 1, we computed $s = n + p$ derivatives with respect to $Z(0)$ and W (the seeding suggested in equation (5) for 60 time steps.

Table 1: Shallow Water Equations Model Problems

Grid Size	n	p
11×11	$3 * 11 * 11 = 363$	4
16×16	$3 * 16 * 16 = 768$	4
21×21	$3 * 21 * 21 = 1323$	4

We compute the sparse Jacobian shown in Figure 4 using SparsLinC or, alternatively, using SparsLinC in the first time step to determine the sparsity structure and then using the compressed Jacobian approach in subsequent time steps. The latter approach is feasible here because the sparsity pattern does not change (continuous use of SparsLinC could accommodate varying sparsity patterns). Table 2 shows the overall runtime of

the black-box ADIFOR approach, the time spent by the intermediate sparsity approach in the computation of the sparse Jacobian using either the SparsLinC or compressed Jacobian approach, and the matrix-matrix accumulation. The total runtime of the intermediate sparsity approach is the sum of the matrix-matrix multiply time and either the SparsLinC or compressed Jacobian entry. Table 3 contains a summary of the memory requirements of these runs (the numbers for the IBM and Sun platforms are very similar).

Table 2: Runtime for Shallow Water Equations Model Problems (in seconds)

	IBM RS/6000			SPARCstation
	11×11	16×16	21×21	11×11
Black-Box ADIFOR	4.24	36.68	71.98	26.63
SparsLinC	4.90	17.77	42.98	12.26
Compressed Jacobian	1.93	8.70	21.51	6.55
Matrix-Matrix Mult.	8.03	38.66	119.32	19.24

Table 3: Memory Requirements for Shallow Water Equations Model Problems (in Mbytes)

	11×11	16×16	21×21
Black-Box ADIFOR	4.70	18.82	53.31
SparsLinC	3.72	13.61	37.82
Compressed Jacobian	3.85	13.84	38.16

We observe that except on the Sun SPARCstation, the intermediate sparsity scheme does not produce any runtime improvements overall, since the matrix-matrix multiplication cost dominates overall cost. In our experiments, we employed a standard sparse matrix-vector multiply kernel written in Fortran as well as an ESSL library routine on the IBM system but found little impact on performance. We also see that the intermediate sparsity scheme requires less memory, since data internal to H require shorter gradients. In all cases, derivatives produced with the various schemes agreed to machine precision.

The disappointing performance of the intermediate sparsity scheme in this particular case is not really surprising. We compute a large number s of derivatives, and the shallow water equations solver is based on a five-point stencil and updates the east and west wind components as well as the geopotential at a cost of only 59 flops per gridpoint. To simulate performance for a computationally more intensive problem, where the computation of the sparse Jacobian corresponding to one time step required more work, we executed the code for one time step from 1 (the original problem) to 16 times, thereby increasing the computational weight of the stencil by a factor of up to 16 while

keeping the structure of the resulting Jacobian unchanged. That is, the floating-point cost of both black-box ADIFOR and the Jacobian associated with a particular time step will increase by the number of repetition, while the cost of the matrix-matrix multiply as well as the memory requirements of the code remains unchanged. The resulting computational behavior is shown in Figures 7 and 8. Speedup here is ratio of CPU time of the black-box ADIFOR approach versus the CPU time of the intermediate sparsity approach.

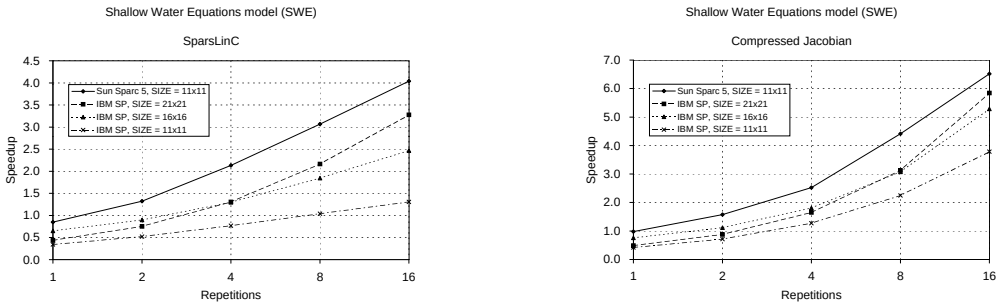

Figure 7: Serial Speedup of Intermediate Sparsity Scheme

Figure 7 shows that we can in fact obtain considerable speedup if the computational weight of the time step update is sufficiently large in comparison with the cost of the matrix-matrix accumulation step. This is also evident in Figure 8, which shows the steadily decreasing influence of the matrix-matrix multiply time as the amount of work per time step increases.

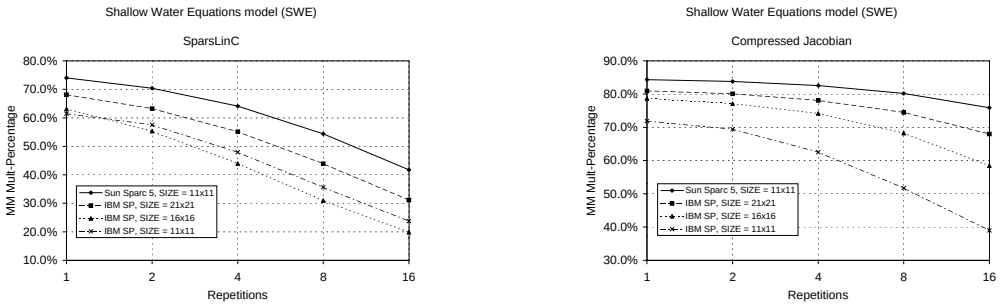

Figure 8: Percentage of Time Spent in Matrix-Matrix Multiply for Intermediate Sparsity Scheme

5 Conclusions

In this article, we considered the differentiation of a leapfrog scheme, a commonly used second-order method for solving differential equations. We employed the associativity of the chain rule to exploit the sparsity of the differential operator at an individual time step and derived a derivative accumulation scheme that combines a derivative code generated by automatic differentiation for the time step update with an explicit matrix-matrix multiply harness that implements the chain rule at a one-timestep level. Experimental results with a shallow water model problem illustrated this approach and also showed that this approach could significantly improve overall performance if the one-timestep update has sufficient computational weight to make up for the cost of the matrix-matrix multiply. We emphasize that the computational harness thus derived is generic — we assumed only that the Jacobian associated with one particular time step is sparse. The coding complexity of the computation occurring in one particular time step is irrelevant, as this part is handled by automatic differentiation. In fact, a more complicated model will likely lead to a more favorable balance between the matrix-matrix multiply and the computation of the Jacobian associated with a particular time step.

Acknowledgments

We thank David Zhi Wang of the Center for Analysis and Prediction of Storms at the University of Oklahoma for providing us with the code for the shallow water equation model.

References

[1] Brett Averick, Jorge Moré, Christian Bischof, Alan Carle, and Andreas Griewank. Computing large sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, 15(2):285–294, 1994.

[2] Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank. *Computational Differentiation: Techniques, Applications, and Tools.* SIAM, Philadelphia, 1996.

[3] Christian Bischof, Ali Bouaricha, Peyvand Khademi, and Jorge Moré. Computing gradients in large-scale optimization using automatic differentiation. Preprint MCS-P488-0195, Mathematics and Computer Science Division, Argonne National Laboratory, 1995. To appear in ORSA Journal of Computing.

[4] Christian Bischof, Alan Carle, George Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Programming*, 1(1):11–29, 1992.

[5] Christian Bischof, Alan Carle, Peyvand Khademi, and Andrew Mauer. ADIFOR 2.0: Automatic differentiation of Fortran 77 programs. *IEEE Computational Science & Engineering*, 3(3):18–32, 1996.

[6] Christian Bischof, Alan Carle, Peyvand Khademi, Andrew Mauer, and Paul Hovland. ADIFOR 2.0 user's guide (Revision C). Technical Memorandum ANL/MCS-TM-192, Mathematics and Computer Science Division, Argonne National Laboratory, 1995. (also CRPC Technical Report CRPC-95516-S).

[7] Christian Bischof, Peyvand Khademi, Ali Bouaricha, and Alan Carle. Efficient computation of gradients and Jacobians by transparent exploitation of sparsity in automatic differentiation. *Optimization Methods and Software*, 7(1):1–39, July 1996.

[8] Christian Bischof, Lucas Roh, and Andrew Mauer. ADIC — An extensible automatic differentiation tool for ANSI-C. Preprint ANL/MCS-P626-1196, Mathematics and Computer Science Division, Argonne National Laboratory, 1996.

[9] Uwe Geitner, Jean Utke, and Andreas Griewank. Automatic computation of sparse jacobians by applying the method of Newsam and Ramsdell. In Martin Berz, Christian Bischof, George Corliss, and Andreas Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 161–172, Philadelphia, 1996. SIAM.

[10] Andreas Griewank and George Corliss. *Automatic Differentiation of Algorithms.* SIAM, Philadelphia, 1991.

[11] Patrick Roache. *Computational Fluid Dynamics.* Hermosa Publishers, 2nd edition, 1996.

[12] Z. Wang, I. M. Navon, X. Zou, and F. LeDimet. A truncated Newton optimization algorithm in meteorology applications with analytic Hessian/vector products. *Computational Optimization and Applications*, 4:241–262, 1995.

[13] Zhi Wang. Variational data assimilation with 2-D shallow water equations and 3-D FSU global spectral models. Technical Report FSU-SCRI-93T-149, Florida State University, Department of Mathematics, December 1993.